Introduction to SQL

Class 13

Course Overview

- Introduction to SQL
  - Databases, Tables
  - Classification of SQL – DDL, DML, DCL, TCL
    - DDL – CREATE, ALTER, DROP
    - DML – SELECT, INSERT, UPDATE, DELETE
    - DCL – GRANT, REVOKE
    - TCL – COMMIT, ROLLBACK, SAVEPOINT
  - Data types, Operators
  - Keys – Primary, Foreign, Composite, Unique, Alternate
  - Integrity Constraints – Domain Integrity Constrains, Entity Integrity Constraints, Referential Integrity Constraints
  - Joins – Outer Joins, Left Outer Joins, Right Outer Joins, Inner Joins.
  - Queries, Subqueries, Functions, Flow Control (IF, CASE, WHILE, REPEAT, LOOP), ,Stored functions ,Stored Procedures
  - Views
  - Indexes, Cursors, Triggers, Events
  - Concurrency and locking (Implicit locks, explicit locks, row level locks, table level locks, database level locks)
  - Tuning SQL queries and optimizing performance
  - SQL Databases vs NoSQL Databases
  - ACID, CAP
  - How SQL databases internally works

Views

SQL CREATE VIEW Example:

- SQL creates a view that shows all customers from Brazil:

```
CREATE VIEW [Brazil Customers] AS
SELECT CustomerName, ContactName
FROM Customers
WHERE Country = 'Brazil';
```

- Query the view above as follows:

```
SELECT * FROM [Brazil Customers];
```

- SQL creates a view that selects every product in the "Products" table with a price higher than the average price:

```
CREATE VIEW [Products Above Average Price] AS
SELECT ProductName, Price
FROM Products
WHERE Price > (SELECT AVG(Price) FROM Products);
```

- Query the view above as follows:

```
SELECT * FROM [Products Above Average Price];
```

SQL Updating a View

- A view can be updated with the CREATE OR REPLACE VIEW statement.
  - ***CREATE OR REPLACE VIEW view_name AS***
    SELECT column1, column2, ...
    FROM table_name
    WHERE condition;

  - *Example*- SQL adds the "City" column to the "Brazil Customers" view:
    - ***CREATE OR REPLACE VIEW [Brazil Customers] AS***
      SELECT CustomerName, ContactName, City
      FROM Customers
      WHERE Country = 'Brazil';

There are certain conditions needed to be satisfied to update a view. If any one of these conditions is **not** met, then we will not be allowed to update the view.

- The SELECT statement which is used to create the view should not include GROUP BY clause or ORDER BY clause.
- The SELECT statement should not have the DISTINCT keyword.
- The View should have all NOT NULL values.
- The view should not be created using nested queries or complex queries.
- The view should be created from a single table. If the view is created using multiple tables then we will not be allowed to update the view.

- The **WITH CHECK OPTION** clause in SQL is a very useful clause for views. It is applicable to a updatable view. If the view is not updatable, then there is no meaning of including this clause in the CREATE VIEW statement.
- The **WITH CHECK OPTION** clause is used to prevent the insertion of rows in the view where the condition in the WHERE clause in CREATE VIEW statement is not satisfied.
- If we have used the **WITH CHECK OPTION** clause in the CREATE VIEW statement, and if the UPDATE or INSERT clause does not satisfy the conditions then they will return an error.
- Example -

CREATE VIEW SampleView AS

SELECT S_ID, NAME

FROM StudentDetails

WHERE NAME IS NOT NULL

WITH CHECK OPTION;

SQL Dropping View

- A view is deleted with the DROP VIEW statement.

  - ***DROP VIEW view_name;***

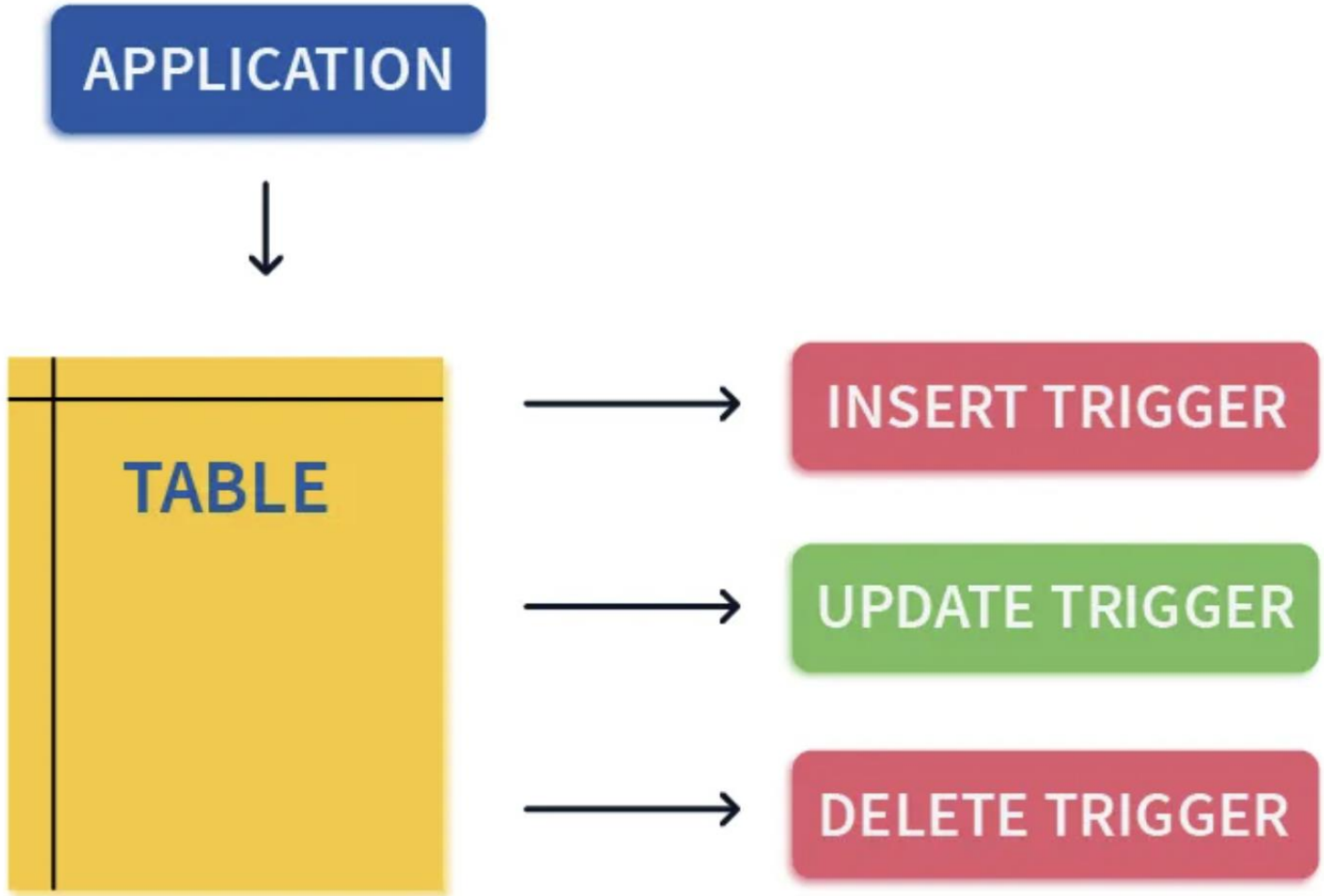
  - *Example*- SQL drops the "Brazil Customers" view:
    - ***DROP VIEW [Brazil Customers];***

Uses of View

- Restricting data access
- Hiding data complexity
- Simplify commands for the user
- Store complex queries
- Rename Columns
- Multiple view facility

Triggers

- A trigger is a database object which fires when an event occurs in a database. For example, a trigger can be set on a record insert in a database table.
- Trigger points of a trigger
  - When any DDL operation is done. E.g., CREATE, ALTER, DROP
  - For a DML operation. e.g., INSERT, UPDATE, DELETE.
  - For a database operation like LOGON, LOGOFF, STARTUP, SHUTDOWN or SERVERERROR


APPLICATION
TABLE
INSERT TRIGGER
UPDATE TRIGGER
DELETE TRIGGER

CREATE Trigger

CREATE [OR REPLACE] TRIGGER trigger_name

{BEFORE | AFTER }

{INSERT [OR] | UPDATE [OR] | DELETE}

ON table_name

[FOR EACH ROW]

WHEN (condition)

[trigger_body]

CREATE Trigger Overview

- CREATE [OR REPLACE] TRIGGER trigger_name: In the first line, we give the name of the trigger we are creating/updating. Here [trigger_name] is to be replaced by the name you want to give to your trigger.
- {BEFORE / AFTER}: Here, we define when the trigger would run, i.e., before or after the DML operation. For example, if we want a trigger to be executed after insertion to a table, we will write after here.
- {INSERT [OR] / UPDATE [OR] / DELETE}: This is the operation or event we have to define, which will trigger a procedure to be executed. If we want a procedure to run after a deletion happens on the table, then we will consider writing delete here.
- on [table_name]: Here, we have to specify the name of the table on which we are attaching the trigger. SQL will listen to changes on this table.
- [for each row]: This line specifies that the procedure will be executed for each one of the rows present. Here we can set a condition for which rows to be impacted. This part is optional, though; in case we don't use this, the trigger shall convert to a "statement-level" trigger rather than being a "row-level" one, i.e., instead of firing the trigger procedure for each row, it will only execute once for each applicable statement.
- WHEN (condition): Here, we mention some condition basis on which the trigger will run. In the absence of a when condition, you can expect it to run for all the eligible rows. This is very important as this will control which rows the trigger must run.
- [trigger_body]: This is the main logic of what to perform once the trigger is fired. In the previous statements, all we defined is when this trigger will be fired, but here we have to define what to do after the trigger is fired. This is the main execution code.

Example

```
CREATE TABLE student(Id integer PRIMARY KEY, first_name varchar(50), last_name varchar(50), full_name varchar(50));


create trigger student_name
after INSERT
on
student
for each row
BEGIN
  UPDATE student set full_name = first_name || ' ' || last_name;
END;
```

Example

```
/* Create a few records in this table */
INSERT INTO student(id, first_name, last_name) VALUES(1,'Alvaro', 'Morte');
INSERT INTO student(id, first_name, last_name) VALUES(2,'Ursula',
'Corbero');
INSERT INTO student(id, first_name, last_name) VALUES(3,'Itziar', 'Ituno');
INSERT INTO student(id, first_name, last_name) VALUES(4,'Pedro', 'Alonso');
INSERT INTO student(id, first_name, last_name) VALUES(5,'Alba', 'Flores');
```

Display Triggers in SQL

- SHOW TRIGGERS LIKE 'stu%’;
- SHOW TRIGGERS **FROM** database_name **WHERE** search_condition
- SHOW TRIGGERS **FROM** database_name;

Drop Triggers in SQL

DROP TRIGGER [IF EXISTS] [schema_name.]trigger_name;

Advantages of Triggers in SQL

- Automation: Triggers automate certain tasks and actions in the database, making it easier to maintain consistency and enforce business rules.
- Improved Data Integrity: Triggers help enforce data integrity by checking for errors or constraints before data is written to the database, helping to prevent data corruption.
- Improved Data Consistency: Triggers can ensure that data is consistent across multiple tables and enforce referential integrity by automatically updating related data.
- Auditing: Triggers can be used to log changes to the data, providing a record of who made changes, when they were made, and what was changed.
- Reusable Code: Triggers allow for reuse of common logic and can reduce the amount of code required to implement complex rules and processes.
- Performance: Triggers can be used to optimize performance by offloading complex processing to the database, reducing the amount of code required in the application.
- Cleaning events – Post someone executes delete statements, we can run certain cleaning logic

Disadvantages of triggers

- Complexity: Triggers can add complexity to a database, making it more difficult to understand and maintain, especially in large and complex systems.
- Performance: Triggers can slow down database performance, especially if they are poorly designed or if they fire too frequently.
- Debugging: Debugging and troubleshooting can be more difficult with triggers, since they are not always easily visible in the code.
- Testing: Triggers can make testing more complex, since they can affect data in unexpected ways.
- Dependencies: Triggers can create dependencies between different parts of the database, making it more difficult to make changes or perform maintenance.
- Security: Triggers can pose a security risk, especially if they are used to execute arbitrary code or to modify sensitive data.
- Portability: Triggers can be difficult to port to different database management systems, since they are not always supported by all systems or may be implemented differently.

Limitations of trigger

- MySQL triggers do not allow to use of all validations; they only provide extended validations. **For example**, we can use the NOT NULL, UNIQUE, CHECK and FOREIGN KEY constraints for simple validations.
- Triggers are invoked and executed invisibly from the client application. Therefore, it isn't easy to troubleshoot what happens in the database layer.
- Triggers may increase the overhead of the database server.

- The trigger body can access the column's values, which are affected by the DML statement.
- The **NEW** and **OLD** modifiers are used to distinguish the column values **BEFORE** and **AFTER** the execution of the DML statement.
- We can use the column name with NEW and OLD modifiers as **OLD.col_name** and **NEW.col_name**.

Trigger Event	OLD	NEW
INSERT	No	Yes
UPDATE	Yes	Yes
ELETE	Yes	No

Trigger Examples

OLD AND NEW

```
CREATE TABLE employee(
    name varchar(45) NOT NULL,
    occupation varchar(35) NOT NULL,
    working_date date,
    working_hours varchar(10)
);
```

OLD AND NEW

```
INSERT INTO employee VALUES
('Robin', 'Scientist', '2020-10-04', 12),
('Warner', 'Engineer', '2020-10-04', 10),
('Peter', 'Actor', '2020-10-04', 13),
('Marco', 'Doctor', '2020-10-04', 14),
('Brayden', 'Teacher', '2020-10-04', 12),
('Antonio', 'Business', '2020-10-04', 11);
```

OLD AND NEW

```
Create Trigger before_insert_empworkinghours
BEFORE INSERT ON employee FOR EACH ROW
BEGIN
IF NEW.working_hours < 0 THEN SET NEW.working_hours = 0;
END IF;
END //

INSERT INTO employee VALUES
('Alexander', 'Actor', '2020-10-012', -13)
```

MySQL: AFTER DELETE Trigger

- An AFTER DELETE Trigger means that MySQL will fire this trigger after the DELETE operation is executed.

```
CREATE TRIGGER trigger_name
AFTER DELETE
  ON table_name FOR EACH ROW
BEGIN
  -- variable declarations
  -- trigger code
END;
```

MySQL: AFTER DELETE Trigger

```
CREATE TABLE contacts
( contact_id INT(11) NOT NULL AUTO_INCREMENT,
  last_name VARCHAR(30) NOT NULL,
  first_name VARCHAR(25),
  birthday DATE,
  created_date DATE,
  created_by VARCHAR(30),
  CONSTRAINT contacts_pk PRIMARY KEY (contact_id)
);
```

```
DELIMITER //
CREATE TRIGGER contacts_after_delete
AFTER DELETE
  ON contacts FOR EACH ROW
BEGIN
  DECLARE vUser varchar(50);
  -- Find username of person performing the DELETE into table
  SELECT USER() INTO vUser;
  -- Insert record into audit table
  INSERT INTO contacts_audit
  ( contact_id,
    deleted_date,
    deleted_by)
  VALUES
  ( OLD.contact_id,
    SYSDATE(),
    vUser );
END; //
DELIMITER ;
```

MySQL: AFTER INSERT Trigger

An AFTER INSERT Trigger means that MySQL will fire this trigger after the INSERT operation is executed.

```
CREATE TRIGGER trigger_name
AFTER INSERT
   ON table_name FOR EACH ROW
BEGIN
   -- variable declarations
   -- trigger code
END;
```

```
DELIMITER //
CREATE TRIGGER contacts_after_insert
AFTER INSERT
  ON contacts FOR EACH ROW
BEGIN
  DECLARE vUser varchar(50);
  -- Find username of person performing the INSERT into table
  SELECT USER() INTO vUser;
  -- Insert record into audit table
  INSERT INTO contacts_audit
  ( contact_id,
    created_date,
    created_by)
  VALUES
  ( NEW.contact_id,
    SYSDATE(),
    vUser );
END; //
DELIMITER ;
```

MYSQL EVENTS

- In MySQL, an event is a named database object that represents a task that runs automatically at a specific time or interval.
- Events are used to perform tasks such as data maintenance, database optimization, or data analysis.
- MySQL Events are also known as “temporal triggers” because they are triggered by time, not by DML events like normal triggers
- MySQL events are similar to a cronjob on Linux or a task scheduler on Windows.
- MySQL Event Scheduler manages the schedule and execution of Events.
- Here are the key components of an event in MySQL:
  - Name: Each event has a unique name that identifies it in the database.
  - Schedule: The schedule specifies when the event should run. You can specify a specific date and time, or a recurring interval, such as every hour, day, or week.
  - Body: The body of the event contains one or more SQL statements that define the task to be performed. This can include anything from data updates and deletions to complex data analysis operations.
  - Execution Context: The execution context determines which database user the event runs as. This can be the MySQL root user, or a specific database user with limited privileges.
  - Status: The status of an event can be either enabled or disabled. When an event is enabled, it runs automatically according to its schedule. When an event is disabled, it does not run until it is enabled again.

MySQL event scheduler configuration

- MySQL uses a special thread called event scheduler thread to execute all scheduled events. You can view the status of the event scheduler thread by executing the SHOW PROCESSLIST command:
  - SHOW PROCESSLIST;
- If the event scheduler is not enabled, you can set the event_scheduler system variable to enable and start it:
  - SET GLOBAL event_scheduler = ON;

Mysql Events

CREATE EVENT [IF NOT EXISTS] event_name
  ON SCHEDULE time_stamp
  DO event_body;

If the event is a one-time event, use the syntax:

AT timestamp [+ INTERVAL]

If the event is a recurring event, use the EVERY clause:

EVERY interval STARTS timestamp [+INTERVAL] ENDS timestamp [+INTERVAL]

Example

```
CREATE TABLE Data (Name VARCHAR(255), age INT);


CREATE EVENT example_event1 ON SCHEDULE AT
CURRENT_TIMESTAMP + INTERVAL 1 Minute DO INSERT INTO new.Data
VALUES('Rahman', 25);


CREATE EVENT example_event2 ON SCHEDULE AT '2021-02-10
12:09:30' DO INSERT INTO new.Data VALUES('Raju', 30);
```

THE EVERY Clause

CREATE EVENT event_name ON SCHEDULE EVERY interval quantity DO event_body;

CREATE EVENT event_hourly ON SCHEDULE EVERY 1 MONTH DO TRUNCATE TABLE data;

```
CREATE EVENT
event_hourly1
ON SCHEDULE
EVERY 1 MONTH
COMMENT 'This truncates the data table each month'
DO
TRUNCATE TABLE data;
```

Example

```
CREATE TABLE messages (
    id INT PRIMARY KEY AUTO_INCREMENT,
    message VARCHAR(255) NOT NULL,
    created_at DATETIME NOT NULL
);
```

One time execution

```
CREATE EVENT IF NOT EXISTS test_event_01
ON SCHEDULE AT CURRENT_TIMESTAMP
DO
  INSERT INTO messages(message,created_at)
  VALUES('Test MySQL Event 1',NOW());
```

One time execution

SHOW EVENTS FROM databasename;

- But if the event has already completed, it wont show up here. For it to show up we need to modify the query to have ON COMPLETE PRESERVE

- The following statement creates another one-time event that is executed after its creation time 1 minute and not dropped after execution.

```
CREATE EVENT test_event_02
ON SCHEDULE AT CURRENT_TIMESTAMP + INTERVAL 1 MINUTE
ON COMPLETION PRESERVE
DO
  INSERT INTO messages(message,created_at)
  VALUES('Test MySQL Event 2',NOW());
```

Creating a recurring event example

```
CREATE EVENT test_event_03
ON SCHEDULE EVERY 1 MINUTE
STARTS CURRENT_TIMESTAMP
ENDS CURRENT_TIMESTAMP + INTERVAL 1 HOUR
DO
  INSERT INTO messages(message,created_at)
  VALUES('Test MySQL recurring Event',NOW());
```

MySQL DROP EVENT Statement

- DROP EVENT [IF EXIST] event_name;

INDEX

- An index is a data structure that allows us to add indexes in the existing table.
- It enables you to improve the faster retrieval of records on a database table
- It creates an **entry** for each value of the indexed columns.
- We use it to quickly find the record without searching each row in a database table whenever the table is accessed.
- We can create an index by using one or more **columns** of the table for efficient access to the records.
- When a table is created with a primary key or unique key, it automatically creates a special index named **PRIMARY**
- We called this index as a **clustered index**. All indexes other than PRIMARY indexes are known as a **non-clustered index or secondary index**.

Need for Indexing in MySQL

- Sequential searching
- Special Lookup tables
- Take example of indexes in books

MySQL CREATE INDEX Statement

- Generally, we create an index at the time of table creation in the database.

```
CREATE TABLE t_index(
   col1 INT PRIMARY KEY,
   col2 INT NOT NULL,
   col3 INT NOT NULL,
   col4 VARCHAR(20),
   INDEX (col2,col3)
);
```

- **CREATE INDEX** [index_name] **ON** [table_name] (**column** names)

Index type for different types of storage engines

SN	Storage Engine	Index Type
1.	InnoDB	BTREE
2.	Memory/Heap	HASH, BTREE
3.	MYISAM	BTREE

MySQL Drop Index

- To delete an index from a table, we can use the following query:
  - DROP INDEX index_name ON table_name [algorithm_option | lock_option];
- Deleting an index requires two compulsory things
  - Name of the index that we want to remove
  - Name of the table where the index belongs
- There are two other options that can be specified when we are deleting an index
  - Algorithm_option : This enables us to specify the specific algorithm for removing the index in a table
  - Algorithm [=] {**DEFAULT** | INPLACE | COPY}

Algorithm Options

- The Drop Index syntax supports mainly two algorithms which are INPLACE and COPY.
- COPY
  - This method creates a new table without the index and then copies the data from the original table to the new table.
  - This method can be slower than the in-place method, but it can be useful if you need to drop an index on a large table and you don't want to block other operations on the table
- INPLACE
  - This method drops the index while keeping the table in place.
  - This method can be faster than the copy method, but it can also cause table-level locking, which can block other operations on the table.
- If the algorithm clause is not defined, MySQL uses the inplace algorithm
  - If INPLACE is not supported, it uses the copy algorithm
- In MySQL, if you drop an index that is a part of a unique constraint, the unique constraint will also be dropped
- In MySQL, If you drop an index that is used in a foreign key constraint, the foreign key constraint will also be dropped.

Lock Option

- This clause enables us to control the level of concurrent reads and writes during the index removal.
- The syntax of **lock_option** are as follows:
  - LOCK [=] {**DEFAULT**|NONE|SHARED|EXCLUSIVE}
- SHARED
  - This type of lock is used when a query needs to read data but does not need to modify it.
  - This lock allows other queries to also read the data, but it prevents other queries from modifying the data until the shared lock is released.
- DEFAULT
  - This is the default lock type used when no lock is specified. It will use the lock type that is appropriate for the specific storage engine being used (e.g. InnoDB or MyISAM).
- NONE
  - This type of lock is used when the query does not need to lock any rows. This is useful for queries that only read data, such as SELECT statements.
- EXCLUSIVE
  - This type of lock is used when the query does not need to lock any rows. This is useful for queries that only read data, such as SELECT statements

Example using Algorithm and Lock

- **DROP INDEX** age **ON** student ALGORITHM = INPLACE LOCK = **DEFAULT**;

MySQL Show Indexes

- We can get the index information of a table using the Show Indexes statement.
- SHOW INDEXES FROM table_name;
- SHOW INDEXES IN table_name FROM database_name;
- SHOW KEYS FROM table_name IN database_name;

SHOW INDEX Query

- Table: It contains the name of the table.
- Non_unique: It returns 1 if the index contains duplicates. Otherwise, it returns 0.
- Key_name: It is the name of an index. If the table contains a primary key, the index name is always PRIMARY.
- Seq_in_index: It is the sequence number of the column in the index that starts from 1.
- Column_name: It contains the name of a column.
- Collation: It gives information about how the column is sorted in the index. It contains values where A represents ascending, D represents descending, and Null represents not sorted.
- Cardinality: It gives an estimated number of unique values in the index table where the higher cardinality represents a greater chance of using indexes by MySQL.
- Sub_part: It is a prefix of the index. It has a NULL value if all the column of the table is indexed. When the column is partially indexed, it will return the number of indexed characters.
- Packed: It tells how the key is packed. Otherwise, it returns NULL.
- NULL: It contains blank if the column does not have NULL value; otherwise, it returns YES.
- Index_type: It contains the name of the index method like BTREE, HASH, RTREE, FULLTEXT, etc.
- Comment: It contains the index information when they are not described in its column. For example, when the index is disabled, it returns disabled.
- Index_column: When you create an index with comment attributes, it contains the comment for the specified index.
- Visible: It contains YES if the index is visible to the query optimizer, and if not, it contains NO.

Types of Index

- Clustered Index
- Non Clustered Index

Parameter	Clustered Index	Non-Clustered Index
Definition	A clustered index is a table where the data for the rows are stored. In a relational database, if the table column contains a primary key, MySQL automatically creates a clustered index named **PRIMARY**.	The indexes other than PRIMARY indexes (clustered indexes) called a non-clustered index. The non-clustered indexes are also known as secondary indexes.
Use for	It can be used to sort the record and store the index in physical memory.	It creates a logical ordering of data rows and uses pointers for accessing the physical data files.
Size	Its size is large.	Its size is small in comparison to a clustered index.
Data Accessing	It accesses the data very fast.	It has slower accessing power in comparison to the clustered index.
Storing Method	It stores records in the leaf node of an index.	It does not store records in the leaf node of an index that means it takes extra space for data.
Additional Disk Space	It does not require additional reports.	It requires an additional space to store the index separately.
Type of Key	It uses the primary key as a clustered index.	It can work with unique constraints that act as a composite key.
Contains in Table	A table can only one clustered index.	A table can contain one or more than a non-clustered index.
Index Id	A clustered index always contains an index id of 0.	A non-clustered index always contains an index id>0.

When should indexes be avoided?

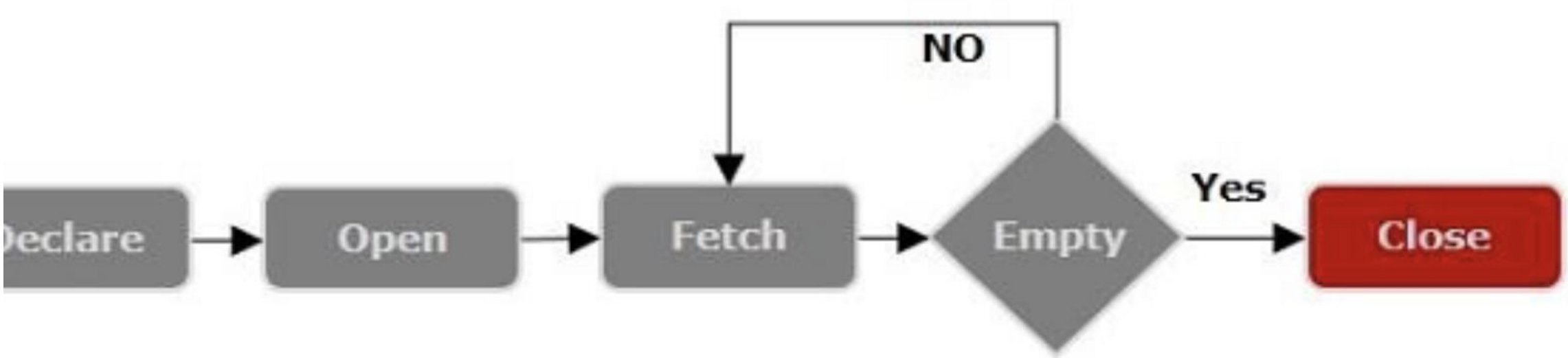
- Indexes should not be used on small tables.
- Tables that have frequent, large batch updates or insert operations.
- Indexes should not be used on columns that contain a high number of NULL values.
- Columns that are frequently manipulated should not be indexed.

MySQL Cursor DECLARE Statement

- A cursor in database is a construct which allows you to iterate/traversal the records of a table.
- In MySQL you can use cursors with in a stored program such as procedures, functions etc.
- In other words, you can iterate though the records of a table from a MySQL stored program using the cursors.
- The cursors provided by MySQL are embedded cursors. They are −
  - READ ONLY − Using these cursors you cannot update any table.
  - Non-Scrollable − Using these cursors you can retrieve records from a table in one direction i.e., from top to bottom.
  - Asensitive − These cursors are insensitive to the changes that are made in the table i.e. the modifications done in the table are not reflected in the cursor. Which means if we have created a cursor holding all the records in a table and, meanwhile if we add some more records to the table, these recent changes will not be reflected in the cursor we previously obtained.

To use a cursor, you need to follow the steps given below (in the same order)

- Declare the cursor using the *DECLARE* Statement.
- Declare variables and conditions.
- Open the declared cursor using the *OPEN* Statement.
- Retrieve the desired records from a table using the *FETCH* Statement.
- Finally close the cursor using the *CLOSE*statement.



The DECLARE Statement

- Using the DECLARE statement you can declare a cursor and associate It with the SELECT statement which fetches the desired records from a table.
- This SELECT statement associated with a cursor does not allow INTO clause.
- Once you declare a cursor you can retrieve records from it using the FETCH statement.
- You need to make sure the cursor declaration precedes handler declarations
- DECLARE cursor_name CURSOR FOR select_statement;

Examples

```
CREATE TABLE tutorials (
   ID INT PRIMARY KEY,
   TITLE VARCHAR(100),
   AUTHOR VARCHAR(40),
   DATE VARCHAR(40)
);

insert into tutorials values(1, 'Java', 'Krishna', '2019-09-01');
insert into tutorials values(2, 'JFreeCharts', 'Satish', '2019-05-01');
insert into tutorials values(3, 'JavaSprings', 'Amit', '2019-05-01');
insert into tutorials values(4, 'Android', 'Ram', '2019-03-01');
insert into tutorials values(5, 'Cassandra', 'Pruthvi', '2019-04-06');
```

Example

```
CREATE TABLE backup (
   ID INT,
   TITLE VARCHAR(100),
   AUTHOR VARCHAR(40),
   DATE VARCHAR(40)
);
```

Following procedure backups the contents of the tutorials table to the backup table using cursors

```
DELIMITER //
CREATE PROCEDURE ExampleProc()
  BEGIN
    DECLARE done INT DEFAULT 0;
    DECLARE tutorialID INTEGER;
    DECLARE tutorialTitle, tutorialAuthor, tutorialDate VARCHAR(20);
    DECLARE cur CURSOR FOR SELECT * FROM tutorials;
    DECLARE CONTINUE HANDLER FOR NOT FOUND SET done = 1;
    OPEN cur;
    label: LOOP
    FETCH cur INTO tutorialID, tutorialTitle, tutorialAuthor, tutorialDate;
    INSERT INTO backup VALUES(tutorialID, tutorialTitle, tutorialAuthor, tutorialDate);
    IF done = 1 THEN LEAVE label;
    END IF;
    END LOOP;
    CLOSE cur;
  END//
DELIMITER ;
```

Connect with me on LinkedIn

Please subscribe to our YouTube channel

Check out my GitHub profile

Follow me on Twitter(X)

Thank you